**METHODOLOGY**

1. **Asking the user for input of a positive integer which defines how many vertices |V| an undirected graph G has and the full path including the full name of a CSV text file that holds the information of the edges set of the undirected graph G.**

The program will ask the user to type a String representing input of a positive integer which defines how many vertices |V| an undirected graph G has and the full path including the full name of a CSV text file that holds the information of the edges set of the undirected graph G. This will be the Pre-Condition of running the program. If the user made a mistake in the input or did not enter a valid input, then the program will crash. The user will just then have to restart the program again.

Once the logical expression is typed-in, the input will be recorded as a String, namely:

1. Vertices
2. pathFile
3. **Reading the set of edges in the CSV text file**

After obtaining the values for the string variable as this will be the Pre-Condition for reading the set of edges in the CSV text file, we do the following:

* 1. Using the pathFile we pass it in as parameters to a new File Class to create a File Class variable file.
  2. Using the Scanner, we then read data that the file contains wit ha while loop and store these data as a String which will then be stored in an Array of Strings.
  3. Once we all have the data from the CSV file to an array of Strings, we can then convert the array of Strings into and array of Integers so that it will be easier for us to analyze.
  4. Post-Condition for this is: the algorithm will return an ArrayList that contains the array of edges namely, listOfEdges.

1. **Creation of the Adjacent Matrix and List**

Next step will be: is to create the adjacent Matrix and adjacent List of the graph of G. Pre-Condition for this is:

1. the listOfEdges which is an Array of an Array of Integers and
2. the dimension of the matrix which is the number of vertices the user typed-in.
3. First we, initialize the 2D array which will represent the Adjacent Matrix of G and the Array of LinkedList of Integers which will represent the Adjacent List of G.
4. Then, we compare the list of edges if the pairs of vertices are within the range of the user specified vertex. If the pair is within range then we modify the adjacent Matrix and List to compensate for the edges included in the graph of G. In the Matrix, we replace a 1 in the location of the edge in the 2D Array. In the List, we link the pairs with each other. Otherwise, if the edges are not within the range of the specified vertex then we do nothing.
5. We repeat step 2 until we iterate over each element in the listOfEdges.
6. Post-Condition for this is: We will have created the modified Adjacent Matrix and Adjacent List for the graph of G.
7. **Searching the most and least popular vertices with their neighbors**

Next, we will search the most and least popular vertices in the graph by using the Adjacency List of G. The pre-condition for this is: the AdjacentList of G.

1. To get the most popular, we pass in as parameters the adjacentList of G in the method ‘getMostPopular’.
2. In the method, we make the first element in the list as the maximum size the list may have. Then we compare the sizes of each element to find out what really is the maximum size that the list has. Once we have found it, we store it in the variable maxSize.
3. We then store in an Array of LinkedList of Integers all the elements in the adjacentList that have the maximum size and this will be considered the Most Popular Vertices in G.
4. To get the least popular, we pass in as parameters the adjacentList of G in the method ‘getLeastPopular’.
5. In the method, we make the first element in the list as the minimum size the list may have. Then we compare the sizes of each element to find out what really is the minimum size that the list has. Once we have found it, we store it in the variable minSize.
6. We then store in an Array of LinkedList of Integers all the elements in the adjacentList that have the minimum size and this will be considered the Least Popular Vertices in G.
7. The post-condition is: We will have the most and least popular vertices of G and their neighbors. The values of the preconditions are not changed.
8. **Printing the most and least popular vertices with their neighbors in the console**

Printing the results are very straightforward, we use System.out.println(“Results as String”) to show the user the results. The pre-conditions are the list of most and lesast popular vertices in G and their neighbors.

The post-condition is: The Results will be printed off on the console for the user to analyze. The values of the preconditions are not changed.

1. **Saves the Adjacent Matrix and Adjacent List of G on the Disk**

Lastly, we will save the Adjacency Matrix and Adjacency List of G in the disk as a CSV text file to the specified location of the user. The pre-condition for this is:

1. the AdjacentList of G.
2. the AdjacentMatrix of G
3. the parent location of the path that the edges were read.
4. To get the parent directory, we use the method file.getParent() of the ListOfEdges CSV text file.
5. Once we have the parent directory, we can then make it the location where to save the AdjacentList and AdjacentMatrix of G.
6. Saving is just like printing the results in the console only this time we use PrintWriter instead of the System.out.println() and StringBuilder instead of the regular String Class. We Only have to format it the way that we are required to.
7. **For the Adjacency Matrix:**
   * 1. the first row of the file starts with the symbol X and then a comma: X,
     2. First row pf the file should also hold the Vertices separated by a comma.
     3. Each row after the first starts with the array row number and includes all elements of the corresponding row of the Adjacency Matrix separated by a comma.
8. **For the Adjacency List:**
   * 1. Each row starts with the number of the vertex and a comma.
     2. After that has all elements of the linked list representing the neighbors of the vertex at the beginning of the row separated by commas.
     3. Vertices should be order in ascending order based on their numbering..
9. The post-condition is: We will have saved the Adjacency List and Matrix of G on the disk. The values of the preconditions are not changed.

**Analysis of Program**

**Let’s analyze the unit cost of the program by calculating the unit cost for each subprogram and the add them all up at the end**

**Structure to Consider: S1 = Asking for the Input, S2 = Reading the set of edges in the CSV text file, S3 = Creation of the Adjacent Matrix and List, S4 = Searching the most and least popular vertices with their neighbors, S5 = Printing the most and least popular vertices with their neighbors in the console ,S6 = Saves the Adjacent Matrix and Adjacent List of G on the Disk**

Let n be the length of the String expression.

**Program S1**

System.***out***.println("Type-in the number of Vertices. ");

String vertices = input.nextLine();

System.***out***.println("Type-in the path of the CSV text file. ");

String pathFile = input.nextLine();

**Then worse case running time for S1 isTs1 = 4 where T represents unit cost**

**Program S2**

**Worst case cost for T2 = 9n2+12n+14**

ArrayList<String> edges = **new** ArrayList<String>();

File file = **new** File(pathFile);

String dirName = file.getParent();

**try**{

Scanner inputStream = **new** Scanner(file);

// hasNext() loops line-by-line

**while** (inputStream.hasNextLine()){

String data = inputStream.nextLine();

edges.add(data);

}

//after loop, close scanner

inputStream.close();

}

**catch** (FileNotFoundException e){

e.printStackTrace();

}

ArrayList<ArrayList<Integer>> listOfEdges = *getEdges*(edges);

**S2a**

ArrayList<ArrayList<Integer>> listOfEdges = **new** ArrayList<ArrayList<Integer>>();

//Iterates for each pairs of vertices

**for** (**int** i=0;i<list.size();i++){

**int** j=0;

//Stores each pair in an array

ArrayList<Integer> edge = **new** ArrayList<Integer>();

String num = "";

**while** (j<list.get(i).length()){

//checks if it is a ','

**if** (list.get(i).charAt(j) == ','){

//convert the string to integer

**int** num1 = Integer.*parseInt*(num);

edge.add(num1);

num = "";

}

//else if the character is the last element

**else** **if** (j == (list.get(i).length()-1)){

num = num + list.get(i).charAt(j);

edge.add(Integer.*parseInt*(num));

}

**else**{

num = num + list.get(i).charAt(j);

}

j++;

}

listOfEdges.add(edge);

}

**return** listOfEdges;

**Program S3**

**Worst case cost for T3 = 5n2+22n+11**

//this is the dimension of the graph of G

**int** dimension = Integer.parseInt(vertices);

//the byte that represent an edge in the graph of G

**byte** one = 1;

//Initialize an array to enable modification later representing the adjacent Matrix

ArrayList<ArrayList<Byte>> adjacentMatrix = getMatrix(dimension);

//Initialize an array that will hold the linked list representing the adjacent list

ArrayList<LinkedList<Integer>> adjacentList = getList(dimension);

//Iterate on each pair of edges

**for** (**int** i=0; i<listOfEdges.size(); i++){

//the edge we are comparing with

ArrayList<Integer> pair = listOfEdges.get(i);

**int** column = pair.get(0);

**int** row = pair.get(1);

//check if it is in range of the specified vertex

**if** ((column < dimension) && (row < dimension)) {

//Modify the Adjacent Matrix

adjacentMatrix.get(column).set(row, one);

adjacentMatrix.get(row).set(column, one);

//Modify the Adjacent List

adjacentList.get(column).add(row);

adjacentList.get(row).add(column);

}

}

**S3a**

ArrayList<ArrayList<Byte>> adjacentMatrix = **new** ArrayList<ArrayList<Byte>>();

//creation of each column

**for**(**int** i=0; i<dimension; i++){

ArrayList<Byte> row = **new** ArrayList<Byte>();

//creation of each row values

**for**(**int** j=0; j<dimension; j++){

**byte** zero = 0;

row.add(zero);

}

adjacentMatrix.add(row);

}

**return** adjacentMatrix;

**S3b**

ArrayList<LinkedList<Integer>> adjacentList = **new** ArrayList<LinkedList<Integer>>();

//creation of each head in the list

**for** (**int** i=0; i<dimension; i++){

LinkedList<Integer> empty = **new** LinkedList<Integer>();

empty.add(i);

adjacentList.add(empty);

}

**return** adjacentList; }

**Program S4**

ArrayList<LinkedList<Integer>> mostPopular = *getMostPopular*(adjacentList);

ArrayList<LinkedList<Integer>> leastPopular = *getLeastPopular*(adjacentList);

**S4a**

ArrayList<LinkedList<Integer>> popular = **new** ArrayList<LinkedList<Integer>>();

//the 1st element in the list will be the compared to find out the maximum size the list has

**int** maxSize = list.get(0).size();

//find out the maximum number of neighbor of the most popular vertex

**for**(**int** i=1; i<list.size(); i++){

**if** (list.get(i).size() > maxSize)

maxSize = list.get(i).size();

}

//System.out.println(maxSize);

//store all the vertices with most neighbors

**for**(**int** j=0; j<list.size(); j++){

**if** (list.get(j).size() == maxSize)

popular.add(list.get(j));

}

**return** popular;

}

**S4b**

ArrayList<LinkedList<Integer>> unpopular = **new** ArrayList<LinkedList<Integer>>();

//the 1st element in the list be compared to find the minimum size the list has

**int** minSize = list.get(0).size();

//find out the minimum number of neighbor of the most popular vertex

**for**(**int** i=1; i<list.size(); i++){

**if** (list.get(i).size() < minSize)

minSize = list.get(i).size();

}

// System.out.println(minSize);

//store all the vertices with the least neighbors

**for**(**int** j=0; j<list.size(); j++){

**if** (list.get(j).size() == minSize)

unpopular.add(list.get(j));

}

**return** unpopular;

**Then the worse case running time for S4 is T4 = 34n+14**

**Program S5**

**Worst case cost for T5 = 18n2+16n+12**

System.***out***.println("Number of Neighbors for MPV: " + (mostPopular.get(0).size()-1) + "\nMPV, Neighbors");

**for**(**int** a=0; a<mostPopular.size(); a++){

System.***out***.print(mostPopular.get(a).get(0));

**for**(**int** e=1; e<mostPopular.get(a).size(); e++){

System.***out***.print("," + mostPopular.get(a).get(e));

}

System.***out***.print("\n");

}

System.***out***.println("Number of Neighbors for LPV: " + (leastPopular.get(0).size()-1) + "\nLPV, Neighbors");

**for**(**int** a=0; a<leastPopular.size(); a++){

System.***out***.print(leastPopular.get(a).get(0));

**for**(**int** e=1; e<leastPopular.get(a).size(); e++){

System.***out***.print("," + leastPopular.get(a).get(e));

}

System.***out***.print("\n");

}

**Program S6**

**Worst case cost for T6 = 19n2+19n+28**

*saveMatrix*(dirName, adjacentMatrix);

*saveList*(dirName, adjacentList);

String matrix = "Adjacent Matrix.csv";

File dir = **new** File (path);

File adjMatrix = **new** File (dir, matrix);

**try** {

PrintWriter pen = **new** PrintWriter(adjMatrix);

StringBuilder sb = **new** StringBuilder();

sb.append("X");

//create the 1st row of the matrix

**for**(**int** q=0; q<adjacentMatrix.size(); q++){

sb.append(',');

sb.append(q);

}

sb.append('\n');

//Create the following rows of the matrix

**for**(**int** n=0; n<adjacentMatrix.size(); n++){

sb.append(n);

//columns of the matrix

**for**(**int** s=0; s<adjacentMatrix.get(n).size(); s++){

sb.append(',');

sb.append(adjacentMatrix.get(n).get(s));

}

sb.append('\n');

}

//write the compiled results

pen.write(sb.toString());

pen.close();

//System.out.println("Writing done");

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

String list = "Adjacent List.csv";

File dir = **new** File (path);

File adjList = **new** File (dir, list);

**try** {

PrintWriter pen = **new** PrintWriter(adjList);

StringBuilder sb = **new** StringBuilder();

//Create the each rows of the list

**for**(**int** n=0; n<adjacentList.size(); n++){

//columns of the list

sb.append(adjacentList.get(n).get(0));

**for**(**int** s=1; s<adjacentList.get(n).size(); s++){

sb.append(',');

sb.append(adjacentList.get(n).get(s));

}

sb.append('\n');

}

//write the compiled results

pen.write(sb.toString());

pen.close();

//System.out.println("Writing done");

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

}

Suppose the program is executed once. Then the cost for all evaluations of test TS1 + TS2 + TS3 + TS4 **= 51n2 + 103n + 83**

Therefore, **51n2 + 103n + 83** <O(n2).

**G. Example using 256 number of vertices**

**RESULT:**

Type-in the number of Vertices.

256

Type-in the path of the CSV text file.

C:\Users\Sidney\workspace\Assignment4\src\GraphEdges.csv

Number of Neighbors for MPV: 35

MPV, Neighbors

21,11,16,22,24,38,49,56,60,79,81,83,88,95,98,121,125,128,135,136,143,146,149,155,162,167,177,184,186,192,202,211,215,219,232,253

Number of Neighbors for LPV: 12

LPV, Neighbors

29,15,67,68,71,88,116,144,163,173,193,206,240

118,1,18,20,72,82,100,117,132,173,177,232,235

**H. Analysis**

The adjacency list has a size of 21.7KB and size on disk of 24 KB while the adjacency matrix has size 129 KB and size on disk is 132 KB. Based on my implementations, the Matrix consumes more ram. The adjacency Matrix has a higher consumption of size because it uses an array to store these values. The edges that is not included in |V| has a Byte value of 0 and edges included has a Byte value of 1 and each of these Bytes consumes 1 byte on the size. So having a dimension of 256x256 would consume already 65.536 KB then the rest will compensate for the values of the row and column number which is of Type Integer which consumes 8 bytes per integer and the String X. The List would have only vertex values included in |V| together with their neighbors of class Integer. There would be 256 Integer values minimum with 8 bytes each which would cost 2.304 KB on size, then the rest would be the number of neighbors in total of all the vertices together with its link or arrow connecting the lists.